Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Data Parallelism: Tecniche Avanzate di Stabilità e Convergenza di modelli su Larga Scala

Francesco Gargiulo, Antonio Francesco Gentile, Emilio Greco

RT- ICAR-NA-2024-06 **Dicembre 2024**

Premessa

In questo rapporto tecnico, illustreremo tecniche avanzate relative al pacchetto DDP nel contesto di addestramento di grandi modelli. Faremo riferimento allo script di test trattato in dettaglio nel rapporto tecnico RT-ICAR-NA-24-04. All'interno dello stesso abbiamo illustrato i principi fondamentali del DDP utilizzando a titolo esemplificativo un classificatore di immagini di abbigliamento partendo dal dataset **Fashion-MNIST** su più GPU.

Questo elaborato tecnico fa parte di una collana di documenti volti a fornire utili informazioni per l'uso efficiente e sostenibile di infrastrutture di calcolo per l'IA basate su architetture multi-GPU. In particolare, questo lavoro è stato reso possibile grazie alle attività realizzate nell'ambito del progetto "Humanities and Cultural Heritage Italian Open Science Cloud – H2IOSC," finanziato dall'Unione europea - NextGenerationEU nell'ambito del PNRR Missione 4, "Istruzione e Ricerca" - Componente 2, "Dalla ricerca all'impresa" - Linea di investimento 3.1, "Fondo per la realizzazione di un sistema integrato di infrastrutture di ricerca e innovazione", decreto di concessione del finanziamento prot. MUR n. 112 del 20-06-2022 (CUP B63C22000730005).

I corsi di formazione forniti da NVIDIA Academy, le competenze interne all'ICAR e la documentazione acquisita attraverso vari canali, hanno permesso di approfondire e applicare tecniche all'avanguardia nel calcolo distribuito consentendo di esplorare e formalizzare modalità operative e casi d'uso delle risorse di calcolo recentemente acquistate dall'ICAR CNR, per il supporto alle ricerche avanzate condotte nell'ambito di infrastrutture per l'AI basate sul paradigma multi-GPU.

1. Introduzione

Negli ultimi anni, l'evoluzione del deep learning ha portato allo sviluppo di modelli sempre più complessi e dalle elevate capacità di generalizzazione. Tuttavia, addestrare questi modelli richiede un'enorme potenza computazionale e l'uso di tecniche di parallelizzazione su larga scala per ridurre i tempi di training. In questo contesto, il pacchetto **Distributed Data Parallel (DDP)** di PyTorch si è affermato come uno strumento essenziale per implementare il parallelismo sui dati, facilitando l'addestramento distribuito su infrastrutture multi-GPU.

Questo rapporto tecnico esplora tecniche avanzate per migliorare la stabilità e la convergenza dell'addestramento di modelli su larga scala, con un focus sull'uso di DDP. Viene discusso l'uso di modelli distribuiti su più processi e lo speed up ottenuto su un numero crescente di GPU, evidenziando i benefici di throughput ottenibili e i limiti di scalabilità che si manifestano con l'aumento delle risorse computazionali. Inoltre, si esaminano tecniche di ottimizzazione avanzate, come Layer-wise Adaptive Rate Clipping (LARC), Layer-wise Adaptive Moments with Blockwise Momentum (LAMB), e NovoGrad, progettate per affrontare le sfide specifiche dell'addestramento distribuito.

Il documento è strutturato come segue: inizialmente, viene presentato un riepilogo delle tecniche fondamentali di parallelizzazione dei dati e delle problematiche legate alla scalabilità. Successivamente, vengono introdotte e analizzate in dettaglio le tecniche di ottimizzazione avanzata, corredate da esempi pratici e scenari applicativi. Infine, il rapporto fornisce una serie di esperimenti che illustrano l'impatto di queste tecniche sulla velocità di convergenza e sulle prestazioni del modello, concludendo con considerazioni pratiche sull'uso delle risorse multi-GPU.

2. Considerazioni sulla scalabilità

Gli esperimenti condotti in RT-ICAR-NA-24-04 hanno mostrato che, all'aumentare del numero di GPU impiegate per parallelizzare l'addestramento di un modello distribuito tra diversi processi worker, ciascuno dei quali utilizza una partizione del dataset, si ottiene un incremento quasi lineare del throughput del processo di addestramento.

Se eseguiamo un test lanciando l'esecuzione dello script di addestramento che fa uso di DDP su un solo nodo:

```
python myscript.py --node-id 0 --num-gpus 4 --num-nodes 1  --epochs 3 --batch-size 512
```

otterremo un output simile al seguente:

```
Epoch =  1: Cumulative Time = 24.508, Epoch Time = 24.508, Images/sec = 2423.3818359375, Validation Loss = 2.483, Validation Accuracy = 0.169
Epoch =  2: Cumulative Time = 48.048, Epoch Time = 23.540, Images/sec = 2501.712646484375, Validation Loss = 0.745, Validation Accuracy = 0.751
Epoch =  3: Cumulative Time = 71.689, Epoch Time = 23.641, Images/sec = 2512.25830078125, Validation Loss = 0.644, Validation Accuracy = 0.763
```

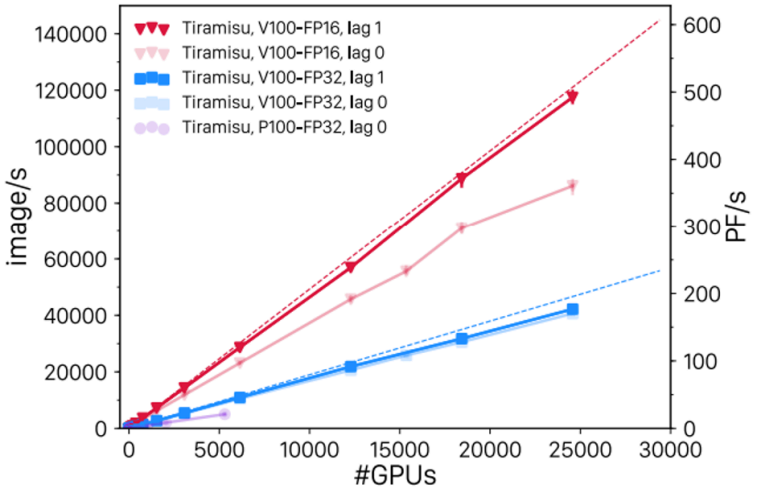
che confrontato con l'output ottenuto attraverso l'esecuzione dell'addestramento su una sola GPU, che riportiamo di seguito:

```
Epoch =  1: Cumulative Time = 91.131, Epoch Time = 91.131, Images/sec = 657.3424213456788, Validation Loss = 0.686, Validation Accuracy = 0.741
Epoch =  2: Cumulative Time = 182.929, Epoch Time = 91.798, Images/sec = 652.560768298225, Validation Loss = 0.612, Validation Accuracy = 0.781
Epoch =  3: Cumulative Time = 274.775, Epoch Time = 91.846, Images/sec = 652.2236577699774, Validation Loss = 0.486, Validation Accuracy = 0.826
Epoch =  4: Cumulative Time = 366.682, Epoch Time = 91.907, Images/sec = 651.7869326969766, Validation Loss = 0.415, Validation Accuracy = 0.852
Epoch =  5: Cumulative Time = 458.566, Epoch Time = 91.884, Images/sec = 651.9506140525963, Validation Loss = 0.378, Validation Accuracy = 0.864
Early stopping after epoch 5
```

possiamo osservare, che il tempo per processare ogni epoca è stato scalato in proporzione al numero di GPU, che nel nostro caso è di 4 unità, così come il numero di immagini processate per secondo (thougput del processo di addestramento).

Passiamo quindi da 652 img/s ad 2500 img/s con un **fattore di scala di 3.8**. La prima osservazione che possiamo fare è che non otteniamo una scalabilità perfettamente lineare, questo perchè interviene una componente significativa del processo di addestramento, dovuta alla comunicazione tra le GPU durante l'aggiornamento dei pesi, che ne riduce la scalabilità.
Oltre alle operazioni di all-reduce di base usate dalla libreria DDP, anche una errata scrittura del codice può introdurre altri elementi di ritardo come ad esempio ulteriori punti di sincronizzazione e

scambio di dati tra i processi ad esempio per eseguire operazioni di I/O per debug o acquisizione di metriche.
Il valore di scalabilità di 3.8 su 4 GPU, per le considerazioni che abbiamo appena fatto è da ritenersi molto buono. Man mano che aumentiamo il numero di GPU, diventa necessario un training multi-nodo e di conseguenza sono necessarie ulteriori considerazioni sull'hardware affinché si abbia una scalabilità efficace. La comunicazione internodo ha caratteristiche diverse da una comunicazione intranodo, in quando in questa si vanno ad aggiungere latenze dovute alle connessioni di rete più importanti.
In linea generale possiamo dire che il throughput cresce in maniera lineare col numero di GPU.
I ricercatori *Kurth at all* in "***Exascale deep learning for climate analytics***" hanno condotto esperimenti di scalabilità aumentando progressivamente il numero di GPU utilizzate, osservando un incremento lineare delle prestazioni fino a raggiungere un punto critico. Durante questi test, l'efficienza parallela della rete di deep learning *Tiramisu* ha mantenuto un'efficienza dell'83,4% su 2048 nodi e del 79,0% su 5300 nodi. Questi esperimenti hanno permesso di comprendere il comportamento della rete su grandi sistemi e di identificare il punto oltre il quale l'incremento di GPU non portava ulteriori miglioramenti in termini di prestazioni.
I ricercatori hanno individuato quindi un punto critico nelle loro prove di scalabilità in cui l'efficienza parallela inizia a diminuire, evidenziando i limiti delle tecniche di parallelizzazione pura. Nei loro test si è notato che le prestazioni scalano in modo quasi lineare fino a 2048 nodi, ma oltre questo limite si osserva una riduzione dell'efficienza parallela al 75,8% a causa della crescente pressione sul sistema di archiviazione dei dati. Dalla loro analisi hanno stabilito che il throughput richiesto dalla rete neurale raggiunge quasi il limite del file system, indicando la necessità di tecniche avanzate di staging dei dati e parallelizzazione del modello per mantenere la scalabilità oltre questo punto critico.



Studi analoghi condotti da *Shallue at all* nel lavoro "***Measuring the effects of data parallelism on neural network training***", hanno evidenziato come, oltre al colli di bottiglia dovuto al sovraccarico di comunicazione e la latenza della rete, vi sono altri fattori da considerare nella scalabilità del processo come la dimensione massima di batch utile alla convergenza. Questo studio evidenzia come esiste una dimensione massima di batch utile alla convergenza che può variare a seconda del

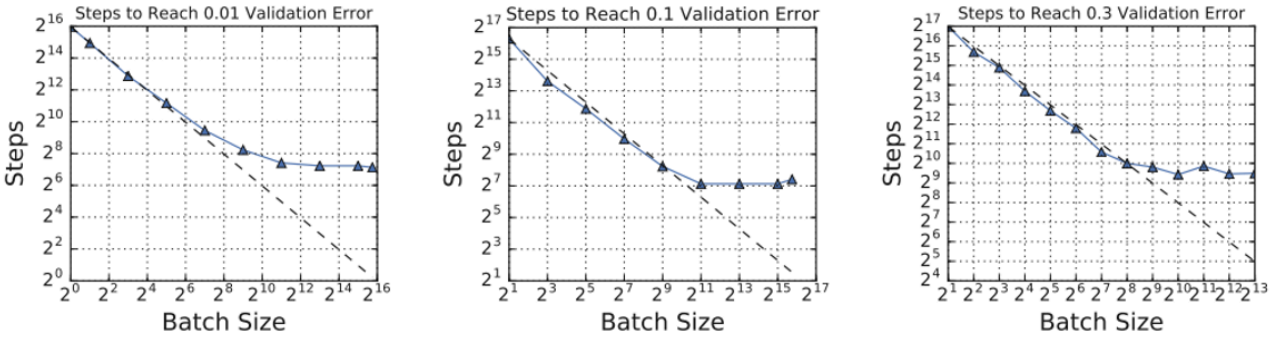
modello e del dataset, evidenziando che non tutte le reti o applicazioni possono scalare allo stesso modo. Ad esempio, nella loro analisi mostrano come reti di tipo ResNet-50 e modelli che usano i Transformer hanno dimostrato di poter sfruttare meglio batch di dimensioni grandi rispetto a modelli meno complessi, il che li rende più stabili rispetto a tecniche di parallelizzazione sui dati.

Questi studi sono utili a comprendere come un aumento della dimensione del batch di dati usato per l'addestramento di un modello può accelerare la convergenza ma con dei limiti. Questo effetto è attribuibile al fatto che, aumentando la dimensione del batch, aumenta anche il contenuto informativo dei dati che contribuiscono al processo di apprendimento; di conseguenza, le stime delle statistiche sui dati come i gradienti diventano più accurate e stabili, favorendo un addestramento più efficace.

In DDP, il dataset viene suddiviso e distribuito su più GPU. Ogni GPU processa un **mini-batch**, che è una parte del batch complessivo. L'idea è che ogni GPU calcola i gradienti per il mini-batch che le è assegnato, e successivamente questi gradienti vengono aggregati per aggiornare i pesi del modello. In questo modo, il batch complessivo può essere visto come la somma dei mini-batch distribuiti sulle diverse GPU.

Per garantire che il modello converga correttamente, i mini-batch non possono essere troppo piccoli. Se ogni GPU riceve un mini-batch troppo ridotto, i gradienti calcolati saranno molto rumorosi e meno rappresentativi della distribuzione dei dati complessiva. Questo effetto può portare a un addestramento instabile o a una lenta convergenza del modello, rendendo il processo meno efficiente.

Quando aumentiamo il numero di GPU, siamo costretti ad aumentare anche la dimensione del batch complessivo, affinché ogni GPU abbia un mini-batch sufficientemente grande da garantire la stabilità dell'addestramento. Tuttavia, questo incremento della dimensione del batch può portare a un **problema di "memoria" nel modello**. Aumentare la dimensione del batch complessivo significa che il modello riceve molte più informazioni contemporaneamente e tende a "memorizzare" i pattern piuttosto che "apprendere" nuove caratteristiche. Questa memorizzazione eccessiva può ridurre la capacità del modello di generalizzare su nuovi dati, portando a un problema di **overfitting** o a una scarsa capacità di adattamento.



(a) Simple CNN on MNIST (b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

Nel lavoro "***ImageNet Training in Minutes***", i ricercatori hanno osservato che la dimensione del batch ha un impatto significativo sull'accuratezza del modello. Quando la dimensione del batch viene aumentata notevolmente, può portare a una degradazione dell'accuratezza di test rispetto a batch più piccoli mantenendo costante il numero di epoche.

In generale, con batch più grandi, il modello ha meno iterazioni per epoca, il che può causare una riduzione della frequenza con cui i pesi vengono aggiornati. Questo implica che, con un batch molto ampio, le capacità di generalizzazione del modello possono diminuire, portando a un effetto noto come "gap di generalizzazione". Questo significa che il modello tende a essere meno capace di adattarsi ai dati di test rispetto a quanto accadrebbe con batch più piccoli.

In questo tipo di analisi i ricercatori hanno dedotto che per mantenere l'accuratezza con batch di grandi dimensioni sono necessarie particolari tecniche avanzate. Alcune di queste, come il ***Linear Scaling Rule*** ed il ***Warmup Scheme***, permettono di aumentare progressivamente il learning rate in modo da compensare la minore frequenza di aggiornamento dei pesi. Pur osservando un netto aumento della stabilità nel processo di parallelizzazione si sono notati dei limiti anche con l'applicazione di queste strategie**.** In conclusione, nonostante ci siano dei progressi nella ricerca nel spingere sempre più avanti l'astina, inevitabilmente ci si scontra coi limiti legati alla dimensione del batch oltre il quale l'accuratezza del modello parallelizzato inizia a degradare significativamente.

Un terzo elemento che impatta fortemente sulle prestazioni del processo di addestramento è **l'ottimizzatore utilizzato**. Alcuni ottimizzatori impattano notevolmente sulla scalabilità poichè utilizzano molta memoria per le loro operazioni di base che di conseguenza sottraggono al modello ed ai dati rallentando quindi le prestazioni dell'intero sistema. Inoltre se non impostati correttamente potrebbero far convergere i modelli distribuiti verso minimi locali allontanando il processo verso la convergenza al minimo globale. Una ricerca che evidenza questo tipo di problema è stata condotta da *Hao Li and all* in "***Visualizing the Loss Landscape of Neural Nets***".

L'idea ed il contributo aggiunto da questo lavoro è stato di migliorare l'ottimizzatore aggiungendo il parametro **momentum** nell'ottimizzazione per accelerare la convergenza e ridurre il rischio di incappare in minimi locali o punti di sella.
I ricercatori di NVIDIA integrano man mano questi importanti passi in avanti nella ricerca, eseguendo upgrade alla libreria DDP.
Per integrare il parametro momentum dello script visto nel lavoro precedentemente possiamo procedere come di seguito:

1. Aggiunta di un argomento ' momentum' , che può essere impostato da riga di comando:

```
parser.add_argument('--momentum', type=float, default=0.9, help='SGD momentum')
```

2. Aggiunta della nuova variabile momentum nei parametri dell'ottimizzatore:

```
optimizer = torch.optim.SGD(model.parameters(), lr=args.base_lr, momentum=args.momentum)
```

3. Stabilità e Convergenza su Larga Scala

Quando ci si approccia a modelli di machine learning su larga scala, spesso si incontrano sfide significative nel mantenere la stabilità e l'efficienza del processo di addestramento. Con l'aumentare delle dimensioni del dataset e la complessità del modello, infatti, risulta necessario aumentare il numero di GPU, il batch size e il learning rate per accelerare il training e raggiungere prestazioni soddisfacenti. Tuttavia, questo processo porta con sé problematiche specifiche, come la divergenza del modello o la difficoltà nel raggiungere un minimo globale. Per mitigare questi problemi, è fondamentale adottare strategie avanzate che permettano al modello di mantenere una traiettoria di apprendimento stabile, migliorando al contempo la velocità di convergenza.

Tra le tecniche attualmente conosciute annoveriamo:

1. **Manipolazione del Learning Rate**: Una tecnica abbastanza semplice per migliorare la convergenza è scalare il learning rate in base alla dimensione del batch. In teoria, quando si aumenta la dimensione del batch di un fattore k, si dovrebbe scalare il learning rate della radice di k per mantenere costante la varianza del gradiente. Tuttavia, nella pratica, si è visto che scalare il learning rate linearmente, cioè di k, funziona meglio in molti casi. Questa tecnica, spesso chiamata ***Linear Scaling***, è comunemente applicata in problemi di machine learning su larga scala.

2. **Warmup del Learning Rate**: In molti casi, un learning rate elevato all'inizio dell'addestramento può portare alla divergenza, con aggiornamenti dei pesi troppo grandi. Per evitare questo problema, alcune ricerche hanno dimostrato che utilizzare una tecnica come il ***Learning Rate Warmup***, porta dei benefici significativi. La tecnica consiste nell'avviare l'addestramento con un learning rate basso, aumentandolo gradualmente fino al valore target nell'arco delle prime epoche. Questo approccio è efficace per garantire che il modello trovi una discesa verso un minimo in modo più stabile.

3. **Batch Normalization (BN)**: La ***batch normalization*** migliora il processo di addestramento riducendo la varianza interna, normalizzando gli input di ogni layer e permettendo di usare un learning rate più alto. Questo metodo è più sofisticato rispetto alla semplice normalizzazione dell'intero dataset e può ridurre la necessità di tecniche come il ***dropout*** per regolarizzare il modello.

4. **Ghost Batch Normalization (GBN)**: Nelle configurazioni con più GPU, BN può essere problematica poiché è complesso calcolare le statistiche del batch in modo centralizzato. La ***ghost batch normalization*** risolve questo problema introducendo rumore addizionale tramite statistiche di mini-batch calcolate su ciascuna GPU, migliorando la generalizzazione su dataset di grandi dimensioni.

5. **Aggiunta di Rumore al Gradiente**: Un altro approccio per migliorare l'ottimizzazione su batch di grandi dimensioni è aggiungere rumore al gradiente. Questo aiuta a mantenere costante la covarianza senza alterare la media, favorendo una convergenza più stabile del modello senza necessità di ridurre eccessivamente il learning rate.

6. **Incremento del Batch Size durante l'addestramento**: Alcuni studi propongono di aumentare la dimensione del batch durante l'addestramento piuttosto che ridurre gradualmente il learning rate. Questo approccio ha mostrato effetti positivi su alcuni modelli, riducendo l'instabilità introdotta da batch troppo piccoli e permettendo un incremento graduale della capacità del modello di generalizzare.

7. **Layer-wise Adaptive Rate Scaling (LARS)**: *LARS* è una tecnica di ottimizzazione che adatta dinamicamente il learning rate per ciascun layer della rete, basandosi sulla norma del gradiente e del parametro di ogni layer. Questa tecnica è molto utile nei contesti di deep learning su larga scala, dove la dimensione del batch è elevata.

8. **Ottimizzatori Avanzati: LARC, LAMB e NovoGrad**:
    - *LARC* utilizza learning rate specifici per ciascun layer, con un clipping per prevenire gradienti troppo grandi.
    - *LAMB* usa learning rate adattivi layer-wise combinati con l'ottimizzatore Adam, risultando particolarmente efficace nei modelli linguistici come BERT.
    - *NovoGrad* calcola medie mobili layer per layer, risultando efficace in vari contesti, dall'elaborazione di immagini a modelli di linguaggio.

Queste tecniche rappresentano strumenti cruciali nell'ottimizzazione di modelli di deep learning, specialmente quando si scala il training su hardware di grandi dimensioni o con batch di grandi dimensioni.

3.1 Manipolazione del Learning Rate e Learning Rate Warm-Up

Incrementare il learning rate è una strategia per contrastare i batch size grandi. Tuttavia, quando il learning rate è eccessivo, può succedere che il modello non converga, con l'accuratezza di validazione che resta molto bassa (intorno a 0.10), indicando che il modello sta praticamente indovinando a caso. Ciò accade quando gli aggiornamenti dei pesi sono così ampi da "saltare" le regioni ottimali.

Nel lavoro di *Alex Krizhevsky*, viene proposta un'euristica per la scelta della dimensione del **batch size** durante l'addestramento di reti neurali convoluzionali, mirata a ridurre gli errori e i problemi legati all'utilizzo di batch troppo grandi.

Krizhevsky propone un approccio di **batch size variabile**:

1. **Batch di grandi dimensioni per i livelli convoluzionali**: Krizhevsky suggerisce di usare batch molto grandi (come 128K) nei livelli convoluzionali, poiché questi livelli contengono la maggior parte del carico computazionale. Questo consente di sfruttare appieno la capacità di parallelizzazione e aumentare l'efficienza.

2. **Batch più piccoli nei livelli fully-connected**: Nei livelli fully-connected, l'euristica prevede l'uso di batch di dimensioni ridotte (es. 128), che permette di ottenere aggiornamenti dei pesi più frequenti e stabili in questa parte del modello. Questo approccio sfrutta una struttura di **parallelismo ibrido**, in cui si utilizza data parallelism nei livelli convoluzionali e model parallelism nei livelli fully-connected.

3. **Aggiustamento del learning rate**: Per rendere il metodo efficace, l'euristica suggerisce di adattare il learning rate proporzionalmente alla dimensione del batch. Krizhevsky trova che, **nella pratica, moltiplicare il learning rate per la stessa quantità con cui si aumenta la dimensione del batch (invece di √k come suggerito dalla teoria) produce risultati migliori**. Questo aiuta a compensare l'effetto di smoothing della discesa del gradiente, mantenendo l'accuratezza.

Questa euristica di batch size variabile aiuta a ridurre il degrado dell'accuratezza che spesso accompagna batch troppo grandi, rendendo il processo di addestramento più efficace senza compromettere eccessivamente la generalizzazione del modello.

Quando si utilizza un batch molto grande, la rete potrebbe divergere all'inizio del processo di apprendimento, soprattutto se il learning rate è impostato troppo alto fin da subito. Questo accade perché, nelle prime epoche, il modello è in una fase di rapido cambiamento, e un learning rate elevato può portare a oscillazioni o a salti troppo ampi nella funzione di perdita. La strategia di warmup proposta in "***Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour***" consiste nell'utilizzare un **learning rate più basso nelle fasi iniziali dell'addestramento** per stabilizzare il processo. In particolare, viene adottata una strategia di **warmup graduale**, in cui il learning rate viene incrementato progressivamente fino a raggiungere il valore desiderato dopo alcune epoche. Questo permette alla rete di adattarsi gradualmente ai grandi batch, evitando la divergenza e facilitando la convergenza verso un minimo globale. La gradualità aiuta anche a evitare che il modello rimanga bloccato in minimi locali o punti di sella, dove i gradienti potrebbero rallentare l'apprendimento.

PyTorch semplifica l'implementazione del warm-up tramite **torch.optim.lr_scheduler.LinearLR**. Utilizzando i parametri consigliati, nelle prime 5 epoche il learning rate cresce linearmente, partendo da un valore iniziale pari al learning rate diviso per il numero di GPU. Ecco come implementare il warm-up nel nostro script visto in RT-ICAR-NA-24-04:

```
# Parse input arguments
parser = argparse.ArgumentParser(description='Fashion MNIST Example',
                        formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--batch-size', type=int, default=32,
                help='input batch size for training')
parser.add_argument('--epochs', type=int, default=40,
                help='number of epochs to train')
parser.add_argument('--base-lr', type=float, default=0.01,
                help='learning rate for a single GPU')
parser.add_argument('--target-accuracy', type=float, default=.85,
                help='Target accuracy to stop training')
```

```
parser.add_argument('--patience', type=int, default=2,
                    help='Number of epochs that meet target before stopping')
parser.add_argument('--num-nodes', type=int, default=1,
                    help='Number of available nodes/hosts')
parser.add_argument('--node-id', type=int, default=0,
                    help='Unique ID to identify the current node/host')
parser.add_argument('--num-gpus', type=int, default=1,
                    help='Number of GPUs in each node')
## TODO momentum: add argument to accept the momentum parameter
parser.add_argument('--momentum', type=float, default=0.9,
                    help='SGD momentum')
## TODO warmup: add argument to accept the warmup epochs parameter
parser.add_argument('--warmup-epochs', type=float, default=5,
                    help='number of warmup epochs')

args = parser.parse_args()
```

Nello script, dopo aver definito l'ottimizzatore, configuriamo il learning rate scheduler per gestire il warm-up:

```
# Define the SGD optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=args.base_lr, momentum=args.momentum)
#Define the scheduler using the torch.optim.lr_scheduler.LinearLR function
scheduler = torch.optim.lr_scheduler.LinearLR(optimizer, start_factor=1/args.num_gpus,
total_iters=args.warmup_epochs)
```

Infine ad ogni epoca nel ciclo di addestramento, occorre richiamare la funzione scheduler.step() per aggiornare il learning rate alla successiva epoca:

```
for epoch in range(args.epochs):

    t0 = time.time()
    train_sampler.set_epoch(epoch)
    train(model, optimizer, train_loader, loss_fn, device)
    ## advance the scheduler to computer the next
    scheduler.step()
    dist.barrier()
    epoch_time = time.time() - t0
    total_time += epoch_time
```

Tuttavia, man mano che la scalabilità del problema aumenta, ossia le dimensioni dei batch e il tasso di apprendimento aumentano, può sorgere un altro problema. Con un **learning rate molto alto**, ogni aggiornamento dei pesi può diventare così grande da superare la magnitudine degli stessi pesi, causando una **divergenza dell'addestramento**. Questo può portare il modello a non riuscire a convergere verso un minimo globale o a presentare un comportamento instabile.

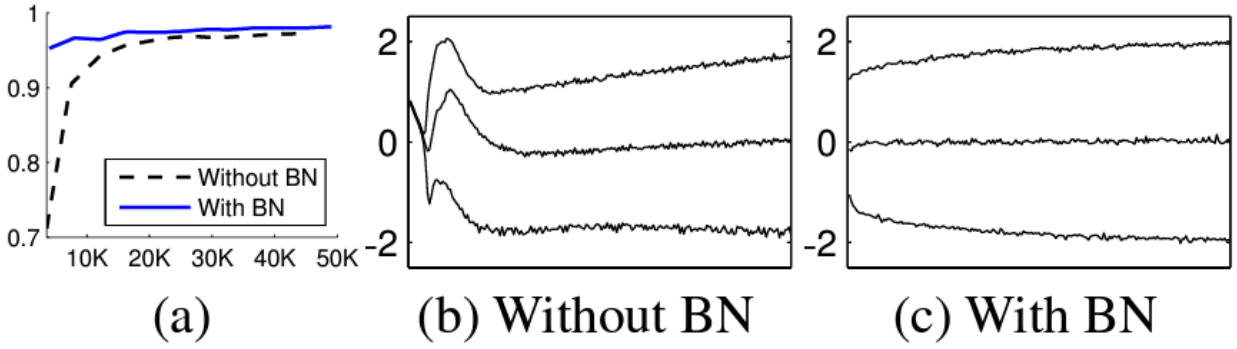
Se il modello diverge, noterai che la **loss potrebbe aumentare invece di diminuire**, e le prestazioni in termini di **accuracy potrebbero peggiorare**. Per risolvere problemi simili su scala più ampia, è spesso consigliabile **combinare tecniche** come il **learning rate warm-up** e il **learning rate decay** per gestire meglio questi aggiornamenti nei primi stadi dell'addestramento, evitando salti troppo ampi nei valori dei pesi del modello.

3.2 Batch Normalization

Nel lavoro di *Ioffe e Szegedy* viene applicata la tecnica chiamata Batch Normalization (BN) che aiuta a migliorare la convergenza nei grandi modelli di deep learning. Durante l'addestramento di reti profonde, la distribuzione degli input a ciascun livello della rete cambia continuamente a causa dell'aggiornamento dei parametri nei livelli precedenti. Questo fenomeno rende più difficile l'addestramento e richiede l'uso di learning rate più bassi e un'attenta inizializzazione dei parametri.

La tecnica di Batch Normalization normalizza l'input di ciascun livello, rendendo la distribuzione degli input più stabile durante l'addestramento. In pratica, BN calcola la media e la varianza degli input all'interno di ciascun mini-batch e utilizza questi valori per normalizzare gli input. Pertanto, normalizzando gli input dei livelli, BN riduce la variazione nei gradienti e **permette l'uso di learning rate più alti senza rischi di divergenza**. Questo accelera notevolmente l'addestramento.

BN ha anche un effetto regolarizzante, riducendo la dipendenza dal dropout in alcune configurazioni. Questo aiuta il modello a generalizzare meglio, migliorando l'accuratezza sui dati di test, allo stesso tempo si riduce la dipendenza dall'inizializzazione dei parametri: Poiché la normalizzazione riduce la variazione nei valori di input, l'addestramento è meno sensibile ai valori iniziali dei parametri, rendendo l'ottimizzazione più stabile e meno soggetta a rimanere bloccata in minimi locali o punti di sella.



3.3 Ghost Batch Normalization

Nel lavoro di *Hoffer et all*., i ricercatori introducono una tecnica denominata Ghost Batch Normalization (GBN) per migliorare la generalizzazione e la convergenza durante l'addestramento con batch di grandi dimensioni su più GPU. Normalmente, la Batch Normalization utilizza le statistiche dell'intero batch per normalizzare gli input ai livelli della rete. Tuttavia, con batch molto grandi distribuiti su più GPU, questo approccio può introdurre problemi di generalizzazione, poiché il batch diventa eterogeneo e le statistiche calcolate possono essere meno rappresentative.

Per risolvere questo problema, i ricercatori propongono di suddividere il grande batch in mini-batch virtuali ("**ghost batches**") all'interno di ciascuna GPU. Ogni mini-batch virtuale calcola le proprie statistiche in isolamento, come se fosse un batch indipendente, il che introduce una lieve variazione nelle statistiche di normalizzazione tra le GPU. In questo modo, GBN evita la necessità di sincronizzare le statistiche su tutte le GPU, riducendo il costo computazionale e migliorando la capacità di generalizzazione.

I benefici introdotti da questa tecnica consistono nel migliorare la generalizzazione: Le statistiche indipendenti per ogni ghost batch introducono una forma di rumore che riduce il rischio di overfitting e favorisce una convergenza verso minimi "piatti" della funzione di perdita, noti per migliorare la generalizzazione. Inoltre dagli studi condotti si è evidenziato un aumentano l'efficienza: GBN permette di sfruttare batch di grandi dimensioni senza la necessità di calcolare le statistiche sull'intero batch distribuito, riducendo così il carico computazionale e la complessità della comunicazione tra GPU.

3.4 Incremento del Batch Size durante l'addestramento

Nel lavoro di *Smith et al.* viene presentato un approccio alternativo alla tradizionale riduzione del learning rate: **aumentare la dimensione del batch durante l'addestramento**. Questa strategia si propone di migliorare la stabilità del processo di apprendimento e la capacità di generalizzazione del modello, specialmente per modelli di grandi dimensioni.

L'idea di base è che, invece di diminuire progressivamente il learning rate per ridurre le fluttuazioni nel gradiente, si può mantenere costante il learning rate e incrementare la dimensione del batch. Questo aumento graduale della dimensione del batch riduce la varianza del gradiente in modo simile alla riduzione del learning rate, ma senza la necessità di impostare parametri di apprendimento più bassi. In pratica l'aumento della dimensione del batch riduce la "scala del rumore" (noise scale) introdotta dall'ottimizzazione stocastica, aiutando il modello a raggiungere minimi più stabili e "piatti" che sono noti per migliorare la generalizzazione. Poiché l'incremento della dimensione del batch consente di ridurre la quantità di aggiornamenti dei parametri, questa tecnica migliora l'efficienza computazionale, permettendo di parallelizzare maggiormente il training e ridurre i tempi complessivi di addestramento. Smith et al. dimostrano che aumentare la dimensione del batch produce effetti simili alla riduzione del learning rate. Quindi, questa tecnica

può essere considerata un'alternativa al tradizionale learning rate decay, consentendo di raggiungere risultati di accuratezza simili sia sul set di training che sul set di test.

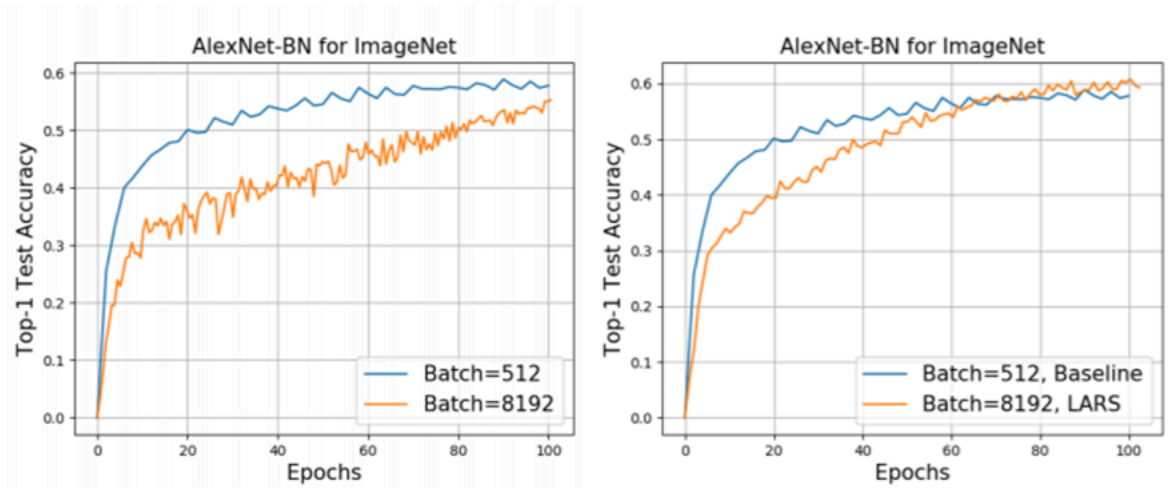
(a) (b)

3.5 Layer-wise Adaptive Rate Scaling (LARS)

Nel lavoro "***Large batch training of convolutional networks with layer-wise adaptive rate scaling***", viene presentata una tecnica chiamata Layer-wise Adaptive Rate Scaling (**LARS**), progettata per migliorare la stabilità e l'efficacia dell'addestramento con batch di grandi dimensioni. LARS è un algoritmo che adatta il tasso di apprendimento su base layer per superare i limiti dell'addestramento tradizionale con Stochastic Gradient Descent (SGD), particolarmente utile quando si aumentano notevolmente le dimensioni del batch.

L'idea di LARS è di **stabilizzare l'addestramento** regolando il learning rate per ogni layer, in base al rapporto tra la norma dei pesi e la norma del gradiente di quel layer. In pratica a differenza di algoritmi come Adam, che adattano il tasso di apprendimento a livello di singolo parametro, LARS applica un tasso di apprendimento diverso per ogni layer della rete. Questo approccio evita l'instabilità che può derivare da un tasso di apprendimento globale troppo alto o troppo basso per alcuni layer. LARS mantiene la magnitudine degli aggiornamenti proporzionale alla norma dei pesi, garantendo che l'aggiornamento non sia troppo grande rispetto ai pesi stessi. Questo aiuta a evitare problemi di divergenza durante la fase iniziale dell'addestramento, in cui i valori dei pesi possono essere piccoli e sensibili a grandi aggiornamenti.

Con LARS, i ricercatori sono riusciti ad addestrare utilizzando un batch fino a 16K per AlexNet e 32K per ResNet-50 senza rilevare una perdita significativa di accuratezza rispetto all'addestramento con batch più piccoli. Tuttavia, hanno osservato che per batch estremamente grandi (come 32K nel caso di AlexNet), è necessario prolungare il numero di epoche di addestramento per recuperare la piena accuratezza, poiché la riduzione del numero di aggiornamenti dei parametri potrebbe influire negativamente sulla convergenza.



(a) Training without LARS (b) Training with LARS

3.6 Ottimizzatori Avanzati: LARC, LAMB e NovoGrad

Gli ottimizzatori avanzati sono algoritmi progettati per migliorare l'efficienza, la stabilità e la velocità di convergenza nei processi di addestramento di modelli di deep learning. Questi algoritmi rispondono alle esigenze crescenti di ottimizzazione per modelli complessi e su larga scala. Di seguito, analizziamo in dettaglio tre ottimizzatori avanzati: **LARC**, **LAMB** e **NovoGrad**, evidenziando le loro caratteristiche, i vantaggi e i contesti d'uso.

LARC (Layer-wise Adaptive Rate Clipping) è un ottimizzatore che si basa sull'adattamento del learning rate specifico per ciascun layer della rete. LARC è stato introdotto per affrontare i problemi di stabilità che possono emergere durante l'addestramento con batch di grandi dimensioni. La caratteristica chiave di LARC è il **clipping del learning rate** in base alla norma dei gradienti di ciascun layer. Questo approccio aiuta a evitare aggiornamenti troppo grandi, che potrebbero portare a instabilità o divergenza durante il training.

L'algoritmo evita esplosioni nei gradienti e stabilizza il processo di ottimizzazione, rendendolo particolarmente utile per reti profonde e batch di grandi dimensioni. La regolazione layer per layer del learning rate consente una convergenza più rapida senza compromettere la stabilità. Questo ottimizzatore è utilizzato principalmente in contesti di addestramento su larga scala, dove l'efficienza e la stabilità sono cruciali, come nei modelli di visione artificiale e nelle reti profonde per l'elaborazione del linguaggio.

LAMB (Layer-wise Adaptive Moments with Blockwise Momentum) è un ottimizzatore sviluppato per grandi modelli come BERT e GPT, utilizzati spesso nel Natural Language Processing (NLP). Esso combina i vantaggi di un **learning rate adattivo layer-wise** con l'ottimizzatore Adam, che utilizza stime dei momenti per migliorare la stabilità e la convergenza. L'algoritmo è stato progettato per supportare batch di dimensioni molto grandi, rendendolo ideale per l'addestramento distribuito su larga scala. L'ottimizzatore calcola il learning rate in base alla norma dei pesi di ciascun layer, come LARC, ma con un approccio specifico per Adam. Sfrutta le tecniche di

ottimizzazione di Adam, calcolando momenti di primo e secondo ordine per ciascun parametro. Questa combinazione consente a LAMB di adattare il learning rate in base alla struttura dei gradienti e alla norma dei pesi.

LAMB ha dimostrato di essere particolarmente efficace su modelli di NLP come BERT, che richiedono stabilità e robustezza nei gradienti per batch molto grandi. Consente di utilizzare batch di dimensioni molto elevate senza compromettere la convergenza, riducendo così i tempi di addestramento per grandi modelli distribuiti. Rispetto agli ottimizzatori standard, LAMB ha dimostrato di ottenere una convergenza rapida senza compromettere l'accuratezza finale del modello.

NovoGrad è un ottimizzatore che calcola i gradienti basandosi su **medie mobili** layer per layer. È progettato per migliorare la convergenza e ridurre la variabilità dei gradienti, risultando efficace in diversi contesti, dall'elaborazione delle immagini ai modelli di linguaggio. NovoGrad combina elementi di RMSProp e Adam, mantenendo una bassa complessità computazionale e un consumo di memoria ridotto. Questo ottimizzatore è adatto a contesti in cui la memoria è limitata, mantenendo una buona stabilità senza consumare molte risorse, dimostrando buone prestazioni sia nei modelli di visione artificiale che nei modelli di linguaggio, risultando versatile e adattabile. La gestione delle medie mobili riduce il rischio di oscillazioni e divergenze durante l'addestramento, migliorando la stabilità del modello. E' impiegato sia in **computer vision** che in **NLP**, offrendo buone prestazioni in ambiti che spaziano dall'elaborazione delle immagini all'analisi dei dati di linguaggio, e risultando efficace anche su modelli distribuiti.

Per modificare il nostro script ed utilizzare questo ottimizzatore i passi da seguire sono i seguenti:

Importare l'ottimizzatore NovoGrad: NovoGrad non fa parte degli ottimizzatori standard di PyTorch, quindi possiamo importarlo dalla libreria *torch_optimizer*, disponibile nel repository Torch Optimizer:

```
import torch_optimizer as opt
```

Successivamente impostiamo l'ottimizzatore al modello: Sostituiamo l'ottimizzatore SGD con NovoGrad, specificando i parametri chiave tra cui il tasso di apprendimento e il parametro grad_averaging. Questo parametro regola la media dei gradienti, pesando una combinazione dei gradienti delle iterazioni attuali e precedenti. È particolarmente utile per problemi complessi come questo, poiché migliora la stabilità della convergenza del modello.

```
optimizer = opt.NovoGrad(model.parameters(), lr=args.base_lr, grad_averaging=True)
```

Conclusioni

Questo rapporto tecnico ha esaminato e illustrato le tecniche avanzate per migliorare la scalabilità e la stabilità dell'addestramento di modelli di deep learning su larga scala utilizzando il pacchetto DDP di PyTorch. Attraverso l'analisi delle prestazioni su GPU multiple, abbiamo evidenziato come l'incremento del numero di GPU porti a un aumento quasi lineare del throughput, sebbene sia inevitabile raggiungere un punto critico oltre il quale la scalabilità si riduce a causa dei costi di comunicazione e sincronizzazione.

Inoltre, abbiamo analizzato l'efficacia degli ottimizzatori avanzati, quali LARC, LAMB e NovoGrad, che consentono di mantenere un processo di addestramento stabile e di migliorare la convergenza anche in presenza di batch di grandi dimensioni. Questi ottimizzatori, applicati con tecniche come il warm-up del learning rate e la batch normalization, hanno dimostrato di essere particolarmente utili per evitare i problemi di divergenza e garantire una traiettoria di apprendimento efficace.

Le tecniche discusse sono strumenti fondamentali per l'ottimizzazione del training su infrastrutture di calcolo ad alte prestazioni. I risultati ottenuti in questo lavoro forniscono un riferimento per l'uso efficiente e sostenibile di risorse di calcolo multi-GPU, contribuendo al progresso delle ricerche avanzate nel campo dell'intelligenza artificiale e supportando le esigenze computazionali dei progetti di ricerca.

Bibliografia

F. Gargiulo, A. Francesco Gentile, E. Greco. (2024, December). *Data Parallelism: Deep Learning con GPU Multiple, in modo efficiente e sostenibile*, RT-ICAR-NA-2024-04 https://intranet.icar.cnr.it/wp-content/uploads/2024/11/RT-ICAR-NA-2024-04.pdf

Kurth, T., Treichler, S., Romero, J., Mudigonda, M., Luehr, N., Phillips, E., ... & Houston, M. (2018, November). Exascale deep learning for climate analytics. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (p. 51). IEEE Press. arXiv:1810.01993

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. arXiv:1811.03600

You, Y., Zhang, Z., Hsieh, C., Demmel, J., & Keutzer, K. (2017). ImageNet training in minutes. arXiv: 1709.05011

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

Keskar, N. S., et al. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv:1609.04836

Li, H., Xu, Z., Taylor, G., & Goldstein, T. (2017). Visualizing the Loss Landscape of Neural Nets. arXiv:1712.09913Hestness, J., et al. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv: 1712.00409

Zoph, Barret, et al. (2017). "Learning transferable architectures for scalable image recognition." arXiv: 1707.07012

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., & Uszkoreit, J. (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., & Keutzer, K. (2014). Densenet: Implementing efficient convnet descriptor pyramids. arXiv preprint arXiv:1404.1869.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*

Li, H., Xu, Z., Taylor, G., & Goldstein, T. (2017). Visualizing the Loss Landscape of Neural Nets. *arXiv:1712.09913*.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. arXiv:1404.5997

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677

Ioffe and Szegedy (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

Smith, S. L., Kindermans, P. J., & Le, Q. V. (2017). Don't Decay the Learning Rate, Increase the Batch Size. arXiv:1711.00489

You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. arXiv:1708.03888